# C212/A592 Lab 7

Intro to Software Systems

## Instructions:

- Review the requirements given below and complete your work. Please compress all files (including main to test your work) into a zip file and submit it through Canvas.
- The grading scheme is provided on Canvas

## Part1: Cannonball

Design a class `Cannonball` to model a cannonball that is fired into the air. A ball has

- An x- and a y-position.
- An x- and a y-velocity.

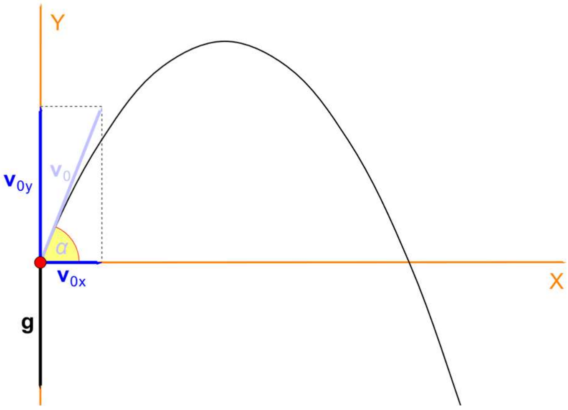



**Figure1.1** initial velocity

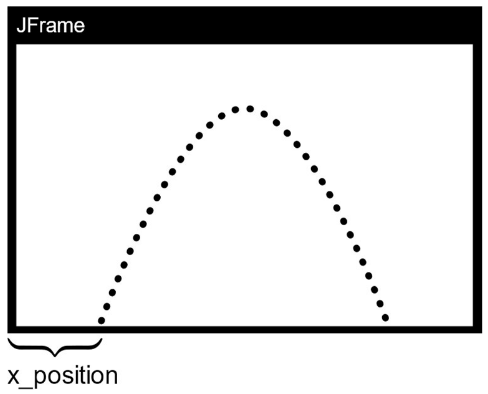



**Figure1.2** expected result

Supply the following methods:

- `Cannonball(double xPosition)`
  A constructor that initializes the x-position field in the class to be the given `xPosition` and y-position to be 0.
- `move(double deltaSec)`
  A method that moves the cannonball to the next position after an interval of `deltaSec` seconds. First compute the distance the ball moves horizontally and vertically in `deltaSec` seconds, using current velocities ($x_{i+1} = x_i + v_x\Delta t$, $y_{i+1} = y_i + v_{y_i}\Delta t - \frac{g\Delta t^2}{2}$), update the x- and y-positions; then update the y-velocity by taking into account the gravitational acceleration of 9.81 $m/s^2$ ($v_{y_{i+1}} = v_{y_i} - g\Delta t$); the x-velocity is unchanged.
- `Point getLocation()`
  A method that returns the current location of the cannonball as a `Point` object, with x- and y-positions rounded to nearest integers.
- `ArrayList<Point> shoot(double alpha, double v, double deltaSec)`
  A method whose arguments are the launch angle $\alpha$, initial velocity $v$ and interval `deltaSec` (in radians). [The initial x-velocity can be computed as $v \times cos\ \alpha$ and the y-

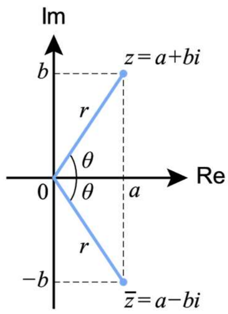
velocity can be computed as $v \times sin\ \alpha$, see **Figure 1.1**]. This method keeps calling `move()` with the given time interval `deltaSec` and adds the new locations as Point objects (using `getLocation()`) to the `ArrayList` until the y-position becomes 0.

Use this class in a driver program that prompts the user for parameters including the starting x-position, the starting angle, the initial velocity and the time interval. Then use these parameters to create a Cannonball object, call **`shoot()`** and use the returned list of positions to **draw the trajectory of the cannonball** in a JFrame similar to what's shown in **Figure 1.2**. Notice that the y coordinates in JComponent start from the top, you need to inverse the y-position when drawing the trajectory to make the y-position starts from the bottom.

**Note**: You are not required to writes Junit tests for the `CannonBall` class.

## Part 2: Complex Numbers

**Figure2** complex plane




Create two classes called `StandardComplex` and `PolarComplex` for performing arithmetic operations with complex numbers. A complex number $z$ have the standard form $z = a + bi$ where $a$ is called the real part, $b$ is called the imaginary part and $i^2 = -1$. Complex numbers can also be written as polar coordinates $(r, \theta)$, where $z = r(cos\theta + isin\theta)$, $r = \sqrt{a^2 + b^2}$ and $\theta = \tan^{-1}\frac{b}{a}$.

For each class,

a) Declare instance variables, constructors, getters and setters
   - Declare private instance variables of type double to represent data `realPart` and `imaginaryPart` in class `StandardComplex`, and `amplitude` and `angle` in class `PolarComplex`.
   - Provide an overloaded constructor that takes arguments to initialize fields in the class.
   - Provide a no-argument constructor that initializes fields to default values.
   - Provide getters and setters for instance variables.

b) Provide the following public methods for each class:
   - `String toString()`
     Prints complex numbers in proper format: (realPart, imageinaryPart) or (amplitude, angle).
   - `Complex getConjugate()`
     Gets the Conjugate of a Complex number: $\bar{z}= a - bi = r(cos\theta - isin\theta)$
     **`Note: Complex`** denotes the corresponding complex number class you are working with
   - `PolarComplex getPolarComplex()`
     Gets Polar Coordinates (i.e., Convert complex number from real and imaginary parts to polar coordinates)
   - `StandardComplex getStandardComplex()`

Gets Complex Number (i.e., Convert polar coordinates to real and imaginary parts of complex number)

c) Provide public methods for each class that perform the following arithmetic operations with another complex number `other`, which can be an instance of a complex number class that is different from **`this`**):

- `boolean isEqual(Object other)`
  Checks if the given input `other` is the same complex number as `this`, regardless of whether `other` is an instance of the same class as `this` or not.
- `Complex add (Object other)`
  Adds two Complex numbers:
  $(a + bi) + (c + di) = (a + c) + (b + d)i$
- `Complex subtract (Object other)`
  Subtract two Complex numbers:
  $(a + bi) - (c + di) = (a - c) + (b - d)i$
- `Complex multiply (Object other)`
  Multiply two Complex numbers:
  $(a + bi) \times (c + di) = (ac - bd) + (ad + bc)i$
- `Complex divide (Object other)`
  Divide two complex numbers.
  $(a + bi) \div (c + di) = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i$
  Divide by zero should not be allowed (You can print a message and throw an exception, something we will discuss later in class)

Write **Junit tests** for methods in part **b)** and **c)**. For part c), you are required to create test cases that use `StandardComplex` objects as input arguments for methods in `PolarComplex` class and vice versa.